LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-658972

An Automated Performance-Aware Approach to Reliability Transformations

J. Lidman, S. A. McKee, D. Quinlan, C. Liao

August 20, 2014

Euro-Par
Porto, Portugal
August 25, 2014 through August 29, 2014

Disclaimer

An Automated Performance-Aware Approach to Reliability Transformations

Jacob Lidman[1], Sally A. McKee[1], Daniel J. Quinlan[2], and Chunhua Liao[2]

[1] Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
{lidman,mckee}@chalmers,
[2] Lawrence Livermore National Laboratory
Livermore, CA, USA
{dquinlan,liao6}@llnl.gov

Abstract. Soft errors are expected to increase as feature sizes shrink and the number of cores increases. Redundant execution can be used to cope with such errors. This paper deals with the problem of automatically finding the number of redundant executions needed to achieve a preset reliability threshold. Our method uses geometric programming to calculate the minimal reliability for each instruction while still ensuring that the reliability of the program satisfies a given threshold. We use this to approximate an upper bound on the number of redundant instructions. Using this, we perform a limit study to find the implications of different redundant execution schemes. In particular we notice that the overhead of higher redundancy has serious implications to reliability. We therefore create a scheme where we only perform more executions if needed. Applying the results from our optimization improves reliability by up to 58.25%. We show that it is possible to achieve up to 8% better performance than Triple Modular Redundancy (TMR). We also show cases where our approach is insufficient.

Keywords: High Performance Computing, Fault Tolerance, N-Modular Redundancy, Reliability Optimization

1 Introduction

Technology trends like shrinking feature sizes and increasing numbers of processor cores on chip make transient faults in hardware increasingly common. These transient faults manifest themselves as bit-flips, and they can originate from external sources (e.g., radiation events) or internal sources (e.g., voltage drops, power supply noise, or leakage). Such faults are termed *soft errors* because they cause no permanent device damage.

As hardware increases in complexity, so does the software that runs on it. Increased complexity results in increased error vulnerability at all levels of the software stack. These problems affect all segments of computing, but they are of particular concern for High Performance Computing (HPC) platforms that

Prepared by LLNL under Contract DE-AC52-07NA27344.

must continue to perform correctly in the presence of such faults. Architects have long proposed hardware enhancements [1] to improve fault tolerance, but implementing dedicated hardware solutions requires design and verification effort, consumes chip real estate, and increases hardware complexity: at some point the proposed solution itself becomes part of the problem. Furthermore, large-scale supercomputers have historically been built from commodity parts not designed for use at such scales; as such, they lack dedicated hardware support, and adding it "after the fact" is difficult and impractical, if not impossible.

Scalable software solutions could become attractive alternatives, but software approaches can incur high performance overheads. As with dedicated fault-tolerant hardware solutions, the fault-tolerant software itself can potentially introduce new errors, since it adds to the (growing) complexity of the application software. To balance these tradeoffs, we study the problem of automatically introducing fault tolerance in software to correct for hardware-induced faults.

Software-implemented hardware fault tolerance (SIHFT) typically relies on adding some form of redundancy. This could entail re-executing instructions when faults are detected (temporal redundancy) or executing multiple independent instructions and adjudicating on their results (spatial redundancy) [15]. The former approach is typically called checkpointing or replay, and HPC systems have traditionally employed it at multiple levels (e.g., in both the OS and the application). Engelmann et. al [2] argue that as the number of nodes increases, system availability will decrease due to single node failures. Increasing the number of nodes also increases the time it takes to save and restore the state when faults occur. The latter approach is termed N-modular redundancy, or NMR (where N signifies the number of independent instructions). In both cases, the amount of work a processing element needs to perform increases, which increases the opportunities for incurring soft errors. An understanding of the tradeoffs between reliability and performance is necessary to make efficient use of hardware resources while delivering a desired level of resilience.

To understand the motivation for automatically deducing N, consider a program in which an instruction I_X consumes the results produced by an instruction I_Y. The probability that I_X produces the correct result can be increased by either hardening I_X or I_Y. In particular, if executing I_X comes at a lower performance cost than executing I_Y, we may prefer to harden the former. In extending to even more instructions, we need to consider how results propagate: producers with more consumers need more protection. The typical approach to applying NMR using a fixed number of redundant executions for all instructions neglects this issue. If I_X and I_Y are redundantly executed different numbers of times, then we also need a means by which to decide how to propagate results when there are fewer consumers than producers (or vice versa). Here we develop an automated approach to determining an appropriate level of redundancy for each instruction and leave the second problem of deciding the interconnection between producers and consumers for future work. We therefore combine producer results before consumer instructions need them.

2 Related Work

Checkpointing has long been the standard approach to hardening HPC applications. Application and system state are periodically saved at dedicated nodes. In the event of a crash, the application is rolled back to a committed state. Lu et. al. [7] highlight the need to keep multiple checkpoint versions to deal with latent errors: this lets them restart in a stable state from before an error was generated, even if the time between occurrence and detection exceeds a single checkpoint interval. The overheads of checkpoint/restart are expected to cause system utilization to decrease rapidly as systems grow to exascale [3]. Researchers are thus investigating a limited amount of modular redundancy, as it allows some soft errors to be handled by a local node cluster, rather than invoking a global restart [3–5]. These approaches use a static number of redundant executions.

Minimizing the performance impact of resilient code is naturally important to the HPC community. Shamsunder et. al. [6] consider the problem of minimizing the number of assertion checks in a multiprocessor environment. They use a CFG-like graph model to represent computation and find an efficient algorithm for the case in which the number of faults is fixed. In contrast, our approach uses a probabilistic description of whether an operation suffers a fault. Misailovic et. al. [8] present an algorithm for replacing operations with less reliable versions while minimizing power consumption and maintaining a preset reliability. Their approach uses a formulation of the optimization problem similar to ours, but they use Integer Linear Programming (ILP) to solve it.

3 Approach

We want to generate code delivering a specified level of reliability while minimizing the performance costs of redundant execution. Our approach is based on the duality between a program's data flow and the probabilistic flow, or how the probability of a state's being correct increases/decreases as it is altered by operations. All data-flow operations increase execution time. Similarly, in the probabilistic interpretation, all non-ideal operations degrade the reliability of producing a correct output. We restrict ourselves to cases where reliability and performance is linearly related. By viewing the operations as constraints on the probabilistic flow, we can find a solution that satisfies a reliability threshold while minimizing performance overheads.

Let a program be represented by a control-flow graph (CFG) $G = \langle V, E \rangle$. Each block $v \in V$ is a sequence of binary $(I_{op}(r_D, r_S, r_T))$ or unary $(I_{op}(r_D, r_S))$ operations over a finite state-space of non-overlapping symbols from a set R. Informally, the semantics of a unary (binary) data-flow operation is that r_D is the destination symbol of an operation I_{op} on symbols r_S (and r_T). We use functions f^V to map symbols r_x to variables that symbolically represent the probability that r_x is correct.

Associated with all unary (binary) data-flow operations is a probability $P_{I_{op}}$ that I_{op} computes correctly given correct operands. Although one would generally expect this probability to change with the operand values, it is common to

assume that the failure of the operation and its operands to be independent [10]. We denote the probability of successful execution prior to optimization as $P_{I_{op}}^{init}$ which we refer to as the *initial execution probability* of I_{op}. The probability that a binary operation I_{op} produces the correct result (for some f^V) is then given by $P_{I_{op}} f^V(r_S) f^V(r_T)$.

3.1 Optimization Problem

Algorithm 1 translates a program into a set of constraints that reflecting the probability that the data flow is correct. The function $newVar()$ returns a fresh variable and *Constraint(op, v_{Id}, f^V, r_D, S, C)* adds a constraint of the form $1 = v_{Id}^{-1} P_{I_{op}} \prod_{s \in S} f^V(s)$ to C and maps r_D to v_{Id} in f^V. We refer to the value of $P_{I_{op}}$ following optimization as the *optimal execution probability* of I_{op}. The $P_{I_{op}}$ variables differ from the v_x variables in that we associate a weight w_{op} with the former. The goal of the optimization process is to minimize $\sum_i w_i P_{I_i}$ subject to the constraints generated by the algorithm: that each variable $\in [0, 1]$ and that the reliability of important symbols reach a predefined level $\hat{P}$. Such an optimization problem is called a *geometric programming problem*.

Algorithm 1: OptGen — Constraint Generator

```
Input: G = ⟨V, E⟩ CFG
Input: f_In^V - Total map from symbol to variable.
Output: C - Set of constraints.
f_Map^V = {v ↦ f_In^V | v ∈ V};
foreach v ∈ V do
    f^V = f_Map^V(v);
    foreach I ∈ v do
        v_Id = newVar();
        case I is
            I_op(r_D, r_S1, r_S2)_l ⇒ Constrain(op, v_Id, f^V, r_D, {r_S1, r_S2}, C) ;
            I_op(r_D, r_S) ⇒ Constrain(op, v_Id, f^V, r_D, {r_S}, C) ;
        f^V(r_D) = v_Id;
```

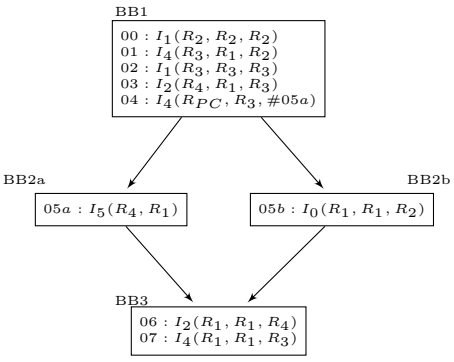
Let $c > 0, x_i \in \mathbb{R}^+$ and $a_i \in \mathbb{R}$ then $g_k(\boldsymbol{x}) = c \prod_i x_i^{a_i}$ is a *mononomial* function. Similarly $f_j(\boldsymbol{x}) = \sum_i g_i(\boldsymbol{x})$ is a *posynomial* function. A geometric programming (GP) problem in standard form is given by:

$$\mathbf{min} f_0(x) \mathbf{s.t.} (\forall j \in [1, m] : f_j(x) \leq 1) \wedge (\forall k \in [1, p] : g_k(x) = 1)$$

GP problems can be transformed into a convex equivalent form by taking the logarithm of all posynomials. The resulting functions are called log-sum-exp functions (i.e., $y = log(\sum_i e_i^x)$). Off-the-shelf solvers [9] can find the minimum of a GP problem in convex form efficiently even for large numbers of variables.

Note that the convex form of a GP problem of only monomial functions, as in our case, is a linear programming problem.

As an example, consider the CFG in Figure 3.1. Variables $\{v_{R_0}, v_{R_1}, v_{R_2}, v_{R_3}, v_{R_4}\}$ denote the probability that symbol R_0, R_1, R_2, R_3 and R_4, respectively, is correct at the entry of a block. The corresponding GP problem is given to the right. We add constraints that symbols that have been updated in the block reach a probability of correctness of at least $\hat{P}$. We find the necessary number of redundant instructions (with initial and optimal execution probabilities p_i and p_o, respectively) by finding the lowest $N = 2n+1$ such that $p_o \leq \sum_{j=n+1}^{N} \binom{N}{j} p_i^j (1-p_i)^{N-j}$, assuming majority voting as the adjudication mechanism.



$$
\begin{aligned}
\textbf{min} \quad & w^T P_{op} \\
\textbf{s.t.} \quad & v_{00} = P_{op(0)} v_{R_2} v_{R_2} \\
& 1 = v_{01}^{-1} P_{op(1)} v_{R_1} v_{00} \\
& 1 = v_{02}^{-1} P_{op(2)} v_{01} v_{01} \\
& 1 = v_{03}^{-1} P_{op(3)} v_{R_1} v_{02} \\
& 1 = v_{04}^{-1} P_{op(4)} v_{02} \\
& 1 \geq v_{01}^{-1} \hat{P} \\
& 1 \geq v_{03}^{-1} \hat{P} \\
& 1 \geq v_{04}^{-1} \hat{P} \\
& 1 = v_{05a}^{-1} P_{op(5a)} v_{R_1} \\
& 1 \geq v_{05a}^{-1} \hat{P} \\
& 1 = v_{05b}^{-1} P_{op(5b)} v_{R_1} v_{R_2} \\
& 1 \geq v_{05b}^{-1} \hat{P} \\
& 1 = v_{06}^{-1} P_{op(6)} v_{R_1} v_{R_4} \\
& 1 = v_{07}^{-1} P_{op(7)} v_{06} v_{R_3} \\
& 1 \geq v_{07}^{-1} \hat{P} \\
& v_* \leq 1 \\
& P_{op(*)} \leq 1 \\
& \forall i : 1 \geq P_{op(i)}^{-1} P_{op(i)}^{init}
\end{aligned}
$$

Fig. 1. Example CFG and Corresponding GP Optimization Problem

3.2 1.mn Voter

The number of redundant executions our approach recommends can be quite high if $\hat{P}$ is high and/or P_{op} is low. As these numbers may represent pessimistic estimates of the number of situations where faults occur, we introduce a voter system that adds redundant executions as needed.

The voting system consists of n stages (for some $n \in \mathbb{N} - \{0\}$). Each stage includes a sequence of redundant executions followed by an assertion that tries to establish whether more redundant executions are needed. The voting system is

shown in pseudo-code in Listing 1.1. In the first stage, we execute $m+1$ versions (from some even $m > 0$). We compare these $m+1$ results using a parity function that is zero iff an even number of operands are ones. If this happens, we vote on the result. Otherwise, we replace the result of the first version with the vote of all results in the first stage and perform m more executions in a second stage. This process then repeats. We do this for a maximum of n stages. We refer to this voter system as the 1.mn voter.

Listing 1.1. 1.mn Voter System

```
x[1] = f(...)
...
x[1+m] = f(...)
IF(PARITY(x[1],...,x[1+m]) = 0)
    res = VOTE(x[1],...,x[1+m])
ELSE {
    x[1] = VOTE(x[1],...,x[1+m])
    x[2] = f(...)
    ...
    x[1+m] = f(...)
    IF(PARITY(x[1],...,x[1+m]) = 0)
        ...
    ELSE
        ...
}
```

4 Evaluation

To evaluate the benefits of our approach, compared to assigning a fixed number of redundant executions, we use six matrix kernels as listed in Table 1. We choose the kernels to show benefits and weaknesses of our approach rather than based on application domain.

Name	Description	Matrix Size
Approx-log	Approximates the logarithm of fix-point numbers in the range [0,1]	5×5
Determinant	Computes the determinant by Laplace expansion recursively	5×5
Gaussian	Implements convolution with a 9×9 Gaussian smoothing kernel	10×10
Bubblesort	Implements Median filtering using 3×3 bubblesort sorting network	10×10
Mincomp	Implements Median filtering using 3×3 minimum comparison sorting network	10×10
Mintime	Implements Median filtering using 3×3 minimum time sorting network	10×10

Table 1. Evaluation Kernels

4.1 Experimental Setup

In our experiments we use the gem5 simulator [14] to model an in-order 32-bit MIPS processor. We modify the simulator to include a pseudo-random number generator for flipping bits in the inputs of the integer ALU in the execution stage. Faults are only injected into instructions that belong to the application. Instructions belonging to the operating system or run-time are ignored. Instructions that do not involve ALU units (e.g., memory load/store or moving data

between registers) are considered ideal. The reason for this assumption is that effective hardware fault-tolerance methods exist for these cases (e.g., redundant buses or error-correcting codes).

The random variable controlling which ALU unit to zap is exponentially distributed with the unit's normalized area and a constant λ_k (i.e., $1-e^{-\lambda_k \frac{Area_{unit}}{Area_{ALU}}}$). The flip probability of each unit is given in Table 3. The initial successful execution probabilities, $P_{I_{op}}^{init}$, similarly use the probability of the ALU unit they use. We vary the constant among experiments to conduct a limit study. Which input bit to flip (of an ALU unit) is uniformly distributed. To obtain area numbers we synthesize an in-house MIPS implementation with the Synopsys Design Compiler using a commercial 65 nm low-power process technology. This implementation models a five-stage MIPS R2000-like processor.

Accessing pages that have not been allocated makes the OS terminate the process. This is a common case when injecting faults [15]. To make sure this does not happen, we change the gem5s virtual memory system so that pages are allocated before we read/write memory. Reading from an invalid memory address then returns a non-deterministic value. This concept has been called *failure-oblivous computing* [16]. With these changes, an invalid memory access can be handled with error-correction logic, as opposed to terminating execution.

4.2 Methodology

We compile our kernels with GCC 4.0.0[3] and optimization level 0 (-O0) to include DWARF debugging information (-g). We analyze the binaries with ROSE [11] to produce a GP problem as described in Section 3. We require that each 32-bit register assigned in the block exit with a probability of at least $\hat{P}$ =0.99[4]. For the sake of demonstration, we use unit weights for all ALU instructions except for div/mul, for which we use 10 (since they are typically about an order of magnitude higher in performance cost). We use CVX 2.0 beta [12, 13] in MATLAB R2012 to solve the GP problem and return the resulting optimal execution probabilities to ROSE. For each pair of initial and optimal execution probabilities we approximate an upper bound on the number of redundant executions, N, for each instruction. Since ROSE::FTTransform [15] operates on source code, we use the DWARF sourceline-to-instruction address mapping to transfer this number back to the source level. Since source lines map to multiple instructions, we use the maximum N of all instructions that map to a particular source line. We use a $1.2N_i$ voter for each source line, where N_i versions were recommended if $N_i > 1$. For $N_i = 1$ we do not apply any transformation. Table 2 shows the values of N for each kernel.

For each kernel we use ROSE::FTTransform to produce two sets of versions. The first set contains the results of applying each kernel to a transformer where we use 3MR, 5MR, and 7MR versions with majority voting. The second set

[3] This particular toolchain is supported by gem5's MIPS simulator in system-call emulation mode.

[4] We arbitrarily choose this value for the purposes of our limit study.

	λ_k					
	1.0		0.1		0.01	
Name	$\widetilde{\mathbf{N}}$	**N Values**	$\widetilde{\mathbf{N}}$	**N Values**	$\widetilde{\mathbf{N}}$	**N Values**
Approx-Log	3	{1,3,7}	3	{1,3}	3	{1,3}
Determinant	3	{1,3,7,9,23}	3	{1,3,5,9}	3	{1,3}
Gaussian	31	{1,3,19,23,27,31}	13	{1,3,9,11,13}	9	{1,3,5,7,9}
Bubblesort	3	{1,3,5}	3	{1,3}	3	{1,3}
Mincomp	7	{1,3,7,9}	5	{1,3,5,7}	3	{1,3,5}
Mintime	7	{1,3,7,9,11}	5	{1,3,5,7}	3	{1,3,5}

Table 2. Distributions of N for All Source Lines of Each Kernel

ALU Unit	**P(flip)**
Add/Sub	1-0.998766
Mul/Div	1-0.953567
Comparator	1-0.999705
And	1-0.999799
Or	1-0.999778
Xor	1-0.999685
Nor	1-0.999852
Shift	1-0.998343

Table 3. Flip probabilities for ALU units

uses a $1.2n$ voter, where n is set to {2,3,4} to produce versions called 1.2-2, 1.2-3, and 1.2-4. These two sets, the set of versions produced by our optimizer (called OPT) and the original kernel (called Orig), collectively make up the kernel versions considered in this work.

For each kernel version we then perform an experiment consisting of 2000 runs with $\lambda_k \in \{1, 0.1, 0.01\}$. We choose the values of λ_k such that reliability results of the evaluations of the original kernel versions map to the range [0%, 100%]. Each run is classified as correct or incorrect. The incorrect class includes runs that time out (meaning the kernel did not complete within one minute[5]), encounter miscellaneous errors (e.g., control-flow errors or invalid syscall requests), and terminate with incorrect results (i.e., silent data corruptions). For each experiment we use the percentage of runs that are classified as correct to measure reliability. Similarly, for each correct run we also compute the median of the number of executed CPU cycles (as reported by gem5) of all runs, which we use to represent performance.

4.3 Results

Table 4 shows the reliability results for each version (rows) and each kernel/λ_k (columns). As expected, results improve with decreasing λ_k. At $\lambda_k = 0.01$, 44.65-84.4% of all runs using the original version complete with correct results. For NMR versions, increasing N does not improve reliability in our evaluations. This is due to the increased fault probability of the majority voter as N increases. The number of clauses for $N = 3$, 5, and 7 is 3, 10, and 33, respectively. The 1.2n versions and the 3MR version thus achieve similar reliabilities. With the exception of the Gaussian kernel, the versions produced by our approach achieve among the best reliabilities for all experiments. We discuss the results of the Gaussian kernel more in Section 5.

Table 5 shows execution performance results. An "-" entry indicates that no execution terminated with correct results. Performance stays somewhat constant over all λ_k for all versions except OPT. This is expected for the original and NMR versions but surprising for the 1.2n versions. These results indicate that we tend to end up in the the same stage independent of λ_k. The fact that performance

[5] Normal program execution takes less then ten seconds for all kernels.

Version	Kernel								
	Approx-Log			Determinant			Gaussian		
	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01
Orig	0%	29.85%	80.6%	0.2%	39.5%	84.4%	0%	0.6%	44.65%
3MR	4.2%	69.3%	92.05%	0.8%	56.4%	87.45%	2.2%	72.15%	92.0%
5MR	0.15%	34.7%	67.6%	0.35%	49.3%	82.4%	0%	36.15%	66.8%
7MR	0%	1.5%	21.8%	0.05%	23.1%	61.9%	0%	1.6%	20.15%
OPT	5.3%	69.9%	91.05%	1.1%	59.0%	90.75%	0%	0%	0%
1.2-2	3.4%	66.9%	90.25%	0.95%	55.9%	88.7%	4.4%	73.1%	91.95%
1.2-3	3.5%	67.4%	90.4%	1.2%	56.9%	89.25%	3.55%	74.95%	92.15%
1.2-4	3.7%	67.55%	90.5%	0.9%	59.2%	90.85%	4.9%	72.85%	91.3%

Version	Kernel								
	Mincomp			Mintime			Bubblesort		
	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01
Orig	0%	21.65%	76.3%	0%	25.25%	76.65%	0%	19.6%	73.2%
3MR	9.95%	79.95%	94.3%	11.45%	79.95%	94.4%	8.65%	77.45%	93.15%
5MR	1.45%	58.75%	83.6%	1.65%	58.25%	82.85%	0.45%	48.65%	77.45%
7MR	0%	18.45%	51.1%	0%	7.65%	38.75%	0%	8.3%	36.8%
OPT	12.95%	79.9%	94.35%	12.3%	79.45%	95.1%	9.9%	76.75%	92.6%
1.2-2	11.6%	81.5%	95.25%	11.45%	79.95%	95.2%	11.15%	75.15%	93.25%
1.2-3	12.95%	80.1%	94.15%	12.2%	80.45%	95.4%	10.85%	77.0%	92.85%
1.2-4	11.75%	78.2%	94.9%	12.2%	79.35%	95.3%	7.7%	75.4%	92.3%

Table 4. Reliability Results for Each Kernel Version at Each λ_k

cost decreases for OPT can be understood by looking at Table 2: the number of executions used decreases for decreasing λ_k. These numbers should not be used to evaluate the performance overhead of redundant execution. Our earlier work [15] shows that we need to hide memory latencies to keep the overhead low. This requires optimizations such as SIMDization and versioning to make use of parallel resources.

Figure 2 shows the geometric means of the reliability and performance results for all runs, excluding those of the Gaussian kernel. For each version, OPT achieves marginally better reliability than the other resilient versions. At the same time, if λ_k is sufficiently low, its performance can even be better then 3MR (which indiscriminately adds redundancy in cases where our approach does not). But for high λ_k the execution overhead is quite high.

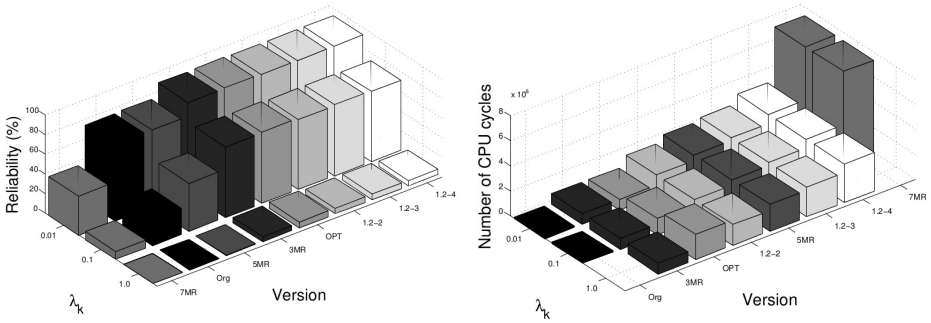


Fig. 2. Geometric Mean of Reliability (left) and Performance (right) Results

Version	Kernel								
	Approx-Log			Determinant			Gaussian		
	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01
Orig	-	89380	89380	64198	63986	63986	-	107258	107192
3MR	226534	225818	225738	107406	106966	106900	313871	308869	308337
5MR	479330	478074	477918	200758	200898	200832	-	1913594	1912978
7MR	-	2031302	2031017	421331	520667	520535	-	6223762	6223072
OPT	358369	219939	219859	419488	197007	101368	-	-	-
1.2-2	399996	398956	398862	159122	158592	158496	1304283	1297262	1296546
1.2-3	576538	575591	575501	207896	207761	207671	3158828	3153470	3152882
1.2-4	750603	750004	749956	255271	256290	256298	4197134	4194096	4193656

Version	Kernel								
	Mincomp			Mintime			Bubblesort		
	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01	λ_k=1.0	λ_k=0.1	λ_k=0.01
Orig	-	124310	124310	-	135194	135194	-	178702	178702
3MR	2250657	2249985	2249901	2250653	2249989	2249901	3689882	3689220	3689136
5MR	6855928	6854958	6854864	6855988	6854958	6854866	9523724	9522746	9522652
7MR	-	21741416	21741266	-	30197146	30197008	-	30197151	30197008
OPT	6959278	4671986	2078021	8224106	5503566	2890332	3727170	3266087	3266005
1.2-2	5231408	5233664	5233950	5247188	5249624	5249954	7188489	7190031	7190215
1.2-3	7674798	7686217	7687520	7714317	7726648	7728076	10515264	10525612	10526772
1.2-4	10053976	10080261	10082966	10131521	10157786	10160536	13786320	13811295	13813971

Table 5. Median CPU Cycles for Each Kernel Version at Each λ_k

5 Discussion

We have introduced an algorithm to minimize the number of redundant executions for instructions in a CFG. This algorithm considers both the reliability of the instruction and its performance cost, and the surrounding framework represents a promising first cut at an automated solution. Nonetheless, we find that our current algorithm can be overly conservative in that it calculates the maximum number of redundant executions, which may, in turn, degrade reliability. For instance, in Section 4, the OPT version of the Gaussian kernel achieves a 0% reliability. This kernel makes heavy use of multiplication, which is our most unreliable operation due to the area of the multiplier.

Looking at Table 4, we see that Gaussian is not very resilient. The reliability at $\lambda_k = 0.01$ is comparable to that of the determinant kernel at $\lambda_k = 0.1$. Table 2 shows that our algorithm deduces that we need as many as 31 replications for most of the statements in this kernel. The 1.mn voter that we introduce to cope with high N does not have the desired effect in this case. Performing redundant execution with just NMR, on the other hand, is not a feasible solution. The overhead of majority voting when N increases is very high (for both performance and reliability). This example highlights the need to consider not just the probability of correct execution but also the uncertainty that we associate with this probability. When the uncertainty becomes too high, we could then include a limited amount of replay (or other construct) to bring down the potential overhead.

We have not considered the case of optimizing performance/reliability over the whole of an acyclic CFG (but rather over each basic block individually). For this case we need to be able to combine contributions from mutually exclusive control-flow paths. An affine join function could be used to achieve this, but then we need to include posynomial equality constraints. Geometric program-

ming does not allow posynomial equality constraints, in general. Allowing these makes it a *signomial programming* [9] problem for which no efficient optimization algorithm is known to exist.

Although we show benefits over 3MR, the improvements are marginal with our current approach. Nonetheless, the fact that our approach considers more variables makes it applicable to more complex environments, for instance where only a subset of the instructions are faulty. We have not investigated such scenarios because the current study has highlighted the importance of considering how the uncertainty of our probabilistic flow calculations increases with the number of instructions in the section of the CFG we are targeting. In particular, we have shown that schemes that only add redundant executions as needed can outperform NMR schemes.

6 Conclusion

One problem in using a redundant execution scheme is how to find the number of executions to use for each instruction. We show an algorithm for approximating this number while taking into account performance costs. Our evaluations show that it is possible to achieve good reliability while still minimizing performance overheads. We improve reliability by up to 58% compared to the original versions of our benchmarks. The median performance cost is up to 8% lower then triple modular redundancy.

Our results vary considerably with the assumed probability of an instruction's successful execution. If this probability is low, we conservatively recommend a high number of redundant versions. Adding more redundancy may be harmful, since it gives the faulty environment more changes to alter the semantics of the execution. Determining this probability is of course a big problem in itself. Although our $1.mn$ voter relieves some of these problems, it does not represent a general solution. Our results however show that reliability/performance improve if we only execute versions as needed to cope with this uncertainty. We believe a better optimization algorithm would factor this uncertainty into decisions.

7 Acknowledgments

The authors thank Alen Bardizbanyan and Kasyab Subramaniyan for helping to synthesize the MIPS processor.

References

1. F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience", *International Journal of High Performance Computing Applications*, 23(4):374–388, Nov. 2009.
2. C. Engelmann, H.H. Ong, and S.L. Scott, "The Case for Modular Redundancy in Large-Scale High Performance Computing Systems", *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, Feb. 2009, pp. 189–194.

3. K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, D. Arnold, "Evaluating the Viability of Process Replication Reliability for Exascale Systems", *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 44:1–44:12.
4. D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, R. Brightwell, "Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing", *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 78:1–78:12.
5. J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, C. Engelmann, "Combining Partial Redundancy and Checkpointing for HPC", *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'12)*, 2012, pp. 615–626.
6. R. Shamsunder, D.J. Rosenkrantz, and S.S. Ravi, "Exploiting Data Flow Information in Algorithm-Based Fault Tolerance", *Proc. International Symposium on Fault-Tolerant Computing (FTCS)*, June 1993, pp. 280–289.
7. G. Lu, Z. Zheng, and A.A. Chien, "When is Multi-version Checkpointing Needed?", *Proc. 3rd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, June 2013, pp. 49–56.
8. S. Misailovic, M. Carbin, S. Achour, Q. Zichao, and M. Rinard, "Reliability-Aware Optimization of Approximate Computational Kernels with Rely", MIT-CSAIL-TR-2014-001, Jan. 2014.
9. S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming", *Optimization and Engineering*, 8(1):67–127, 2007.
10. M. Carbin, S. Misailovic, and M.C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware", *Proc. SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, Oct. 2013, pp 33–52.
11. D. Quinlan and C. Liao, "The ROSE Source-to-Source Compiler Infrastructure", *Cetus Users and Compiler Infrastructure Workshop, with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011.
12. M. Grant and S. Boyd, "CVX: Matlab Software for Disciplined Convex Programming, version 2.0 beta", `http://cvxr.com/cvx`, June 2014.
13. M. Grant and S. Boyd, "Graph Implementations for Nonsmooth Convex Programs", *Recent Advances in Learning and Control (Lecture Notes in Control and Information Sciences 371)*, Springer, 2008, pp. 95–110.
14. N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood, "The Gem5 Simulator", *Computer Architecture News*, 39(2):1–7, May 2011.
15. J. Lidman, D.J. Quinlan, C. Liao, and S.A. McKee, "ROSE::FTTransform – A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research", *Proc. 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, June 2012, pp.1–6.
16. M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee, Jr., "Enhancing Server Availability and Security Through Failure-Oblivious Computing", *Proc. 6th Symposium on Operating Systems Design & Implementation (OSDI)*, Dec. 2004, pp. 21–21.